CS1101: Foundations of Programming

Structures

Dept. of Computer Science & Engineering
Indian Institute of Technology Patna

Structure

- A structure is a collection of one or more variables, possibly of different types, grouped together under a single name for convenient handling
- This is very much useful when the data items are of different types
- Arrays can store similar items only
- Better readability even when the constituent data items are of the same type
- Structures help to organize complicated data
- It permits a group of related variables to be treated as a unit instead of as separate entities.
- Example:
  - Student record: name, roll, height, marks, etc.
  - Complex number: real, imaginary

Defining a structure

The composition of structure is as follows:

```
struct <name of structure> {
   <data type> <member name1>;
   <data type> <member name2>;
   …
   <data type> <member namek>;
};
```

Example:

```
struct student {
   char name[30];
   char roll[10];
   float weight;
   int marks;
};
struct student x, y;
```

Example:

```
struct complex {
   float real;
   float imaginary;
};
struct complex a, b;
```

Structure

- `struct` **is the required C keyword**
- **Do not forget ; at the end of the structure definition**
- **The individual members can be ordinary variables, pointers, arrays, or other structures**
- **The member names within a particular structure must be distinct from one another**
- **A member name can be the same as the name of a variable defined outside the structure**

Structure definition & Structure variable declaration

Structure definition:

```
struct point {
   float xcoord;
   float ycoord;
};
```

No memory is allocated.

Only defining a new data type.

Structure variable declaration:

```
struct point a, b, c;
```

Here `a, b, c` are variables of the type `struct point`.

Memory is allocated for `a, b, c`.

Variable declaration is allowed after definition.

Structure variable declaration

Separately:

```
struct point {
   float xcoord;
   float ycoord;
};
struct point a, b, c;
```

Together:

```
struct point {
   float xcoord;
   float ycoord;
} a, b, c;
```

The struct definition can be reused later like

```
struct point x, y;
```

Another way:

```
struct {
   float xcoord;
   float ycoord;
} a, b, c;
```

In this case we do not have a name for the struct

Hence we cannot reuse the struct definition

The `typedef` construct

- **The `typedef` construct can be used to give new names to (existing) data types in C**

  `typedef float density;` **// density is a new name for float**

  `density d1, d2;` **// d1, d2 are two variables**

  `d1 = 2.4;`

- **In the `typedef` definition no variable is allocated space**

Complicated examples of `typedef`

```
typedef int intarr[50];
intarr    A,      // A is an array of 50 integers
          B[20],  // B is an array of 20 arrays of 50 integers
          *C;     // C is a pointer to an array of 50 integers
int A[50], B[20][50], (*C)[50];

typedef int *intptr;
intptr    p,       // p in an int pointer
          q[100],  // q is an array of 100 int pointers
          *r;      // r is a pointer to an int pointer
int *p, *q[100], **r;
```

Structures and typedef

Without typedef:

```
struct point {
   float xcoord;
   float ycoord;
};
struct point a, b, c;
```

Here `struct point` is new data type.

With typedef:

```
typedef struct {
   float xcoord;
   float ycoord;
} point;
point a, b, c;
```

Here `point` is a new data type. Since `struct` is not followed by a name, this data type can be addressed only by the given name `point`.

Accessing the members of a structure

- **The members of a structure are accessed individually**
- **A member of structure can be accessed in the following manner:** `<variable-name>.<member-name>`
- **The** `variable-name` **is the name of the structure variable,** `member-name` **refers to the name of a member within the structure**
- **Dot operator is used to access members of structure through structure-type variable**

```
struct point {
   float xcoord;
   float ycoord;
} a, b;
a.xcoord = 3.2; b.ycoord = 2.3; a.ycoord = b.xcoord = 0.0;
```

Structure initialization

- **Structure variable may be initialized in the following. It is similar to array initialization.**
- **Values are provided within braces and individual elements will be separated by commas**

```
struct point {
   float xcoord;
   float ycoord;
};
struct point a = {2.0, 1.0};
struct point b = {0,0};
```

```
a.xcoord = 2.0;
a.ycoord = 1.0;
b.xcoord = 0;
b.ycoord = 0;
```

Example: Addition of complex numbers

```
int main(){
   struct complex{
     float real;
     float imag;
   } a, b, c;
   scanf("%f%f",&a.real, &a.imag);
   scanf("%f%f",&b.real, &b.imag);
   c.real = a.real + b.real;
   c.imag = a.imag + b.imag;
   printf("c = %f + j (%f)\n", c.real, c.imag);
   return 0;
}
```

Structure declaration can be outside `main()` as well. This is necessary if a program has multiple functions using a structure type.

Assignment of structure variables

```
struct stud{
    int roll;
    char name[20];
    float marks;
};
```

```
int main(){
    struct stud a = {23, "Kanika", 92.3};
    struct stud b = {47, "Sanjoy", 42.5};
    struct stud c;
    c = b;
    return 0;
}
```

A structure variable can be directly assigned to another variable of similar type. All individual members are get assigned / copied.

But two structures cannot be compared for equality in the following manner

```
if( a == b ) {...}
```

Copying structures

```c
int a[4]={1,2,3,4};
int b[4];
b = a; // not allowed
```

```c
struct list{
   int a[4];
};
struct list x, y;
x.a[0] = 0; x.a[1] = 1;
x.a[2] = 2; x.a[3] = 3;
y = x; // allowed
```

Structures can be copied directly even if it contains array.

Assigning all members together

```
struct stud{
   int roll;
   char name[20];
   float marks;
} x;

x = { 24, "Anushree Pal", 98.3 }; // NOT ALLOWED

x = (struct stud){ 24, "Anushree Pal", 98.3 }; // ALLOWED
```

Size of a structure

```
struct stud {
   char name[34];
   int roll;
   float marks;
} x;
```

Total space required is 34 + 4 + 4 = 42 bytes. However, `sizeof(x)` can return 44 or 48 (nearest larger multiple of 4 or 8).

```
struct stud {
   char *name;
   int roll;
   float marks;
} x;
```

Assuming 64-bit addresses, `sizeof(x)` will return 16 as 8+4+4=16. This will be true irrespective of how much memory you `malloc` to `x.name`.

Arrays of structures

- **Once a structure data type is defined, we can declare arrays of structures**

```
struct stud {
   int roll;
   char name[20];
   float marks;
};
struct stud list[10];
```

- **The individual members can be accessed as**

```
list[5].roll
list[8].name
list[2].name[4]
```

Example: Store a list of students name & CPI

```
struct rec {
   char name[50]; float cpi;
} ;
int main(){
   int i; float avg = 0.0; struct rec stud[100];
   printf("Enter record for 100 students\n");
   for(i=0; i<100; i++){
     printf("Enter name and cpi");
     scanf("%s%f", stud[i].name, &stud[i].cpi);
   }
   for(i=0; i<100; i++) avg += stud[i].cpi;
   avg /= 100;
   printf("Average CPI=%f\n",avg);
   return 0;
}
```

Nested structures

```
typedef struct {
   float x, y;
} point ;

typedef struct {
   point a, b, c;  float area;
} triangle ;

triangle T = {{1,2}, {3.0,4.0}, {-6,-5}, -1};
T.area = T.a.x * (T.b.y - T.c.y) +
         T.b.x * (T.c.y - T.a.y) +
         T.c.x * (T.a.y - T.b.y);
T.area /= 2;
if(T.area < 0) T.area = -T.area;
```

Structure containment cannot be recursive

- It is not allowed to that ‘`struct a`’ contains ‘`struct a`’ as a member
- It is not allowed that ‘`struct a`’ contains ‘`struct b`’ and ‘`struct b`’ contains ‘`struct a`’
- It is allowed that a structure contains pointer to a structure of the same type
- These are known as self-referencing pointers
- Such pointers are used extensively to create chains and other types of linked data structures

```
typedef struct _stud {   // a name after struct is mandatory
  int roll; char name[30]; float cpi;
  struct _stud *next;   // self referencing pointer
} student;
```

Structures and functions

```
typedef struct {
   float real, imag;
} complex ;

void swap(complex a, complex b){
   complex t;
   t = a; a = b; b = t;
}

void print(complex a){
   printf("(%f,%f)\n",a.real, a.imag);
}
```

```
int main(){
   complex a={3, 4};
   complex b={7, 8};
   print(a); print(b);
   swap(a,b);
   print(a); print(b);
   return 0;
}
```

Structures and functions

```
typedef struct {
   float real, imag;
} complex ;

void swap(complex a, complex b){
   complex t;
   t = a; a = b; b = t;
}

void print(complex a){
   printf("(%f,%f)\n",a.real, a.imag);
}
```

```
int main(){
   complex a={3, 4};
   complex b={7, 8};
   print(a); print(b);
   swap(a,b);
   print(a); print(b);
   return 0;
}
```

Output:

```
(3.000000,4.000000)
(7.000000,8.000000)
(3.000000,4.000000)
(7.000000,8.000000)
```

Structures can be returned from functions

```c
typedef struct {
   float real, imag;
} complex ;

complex add(complex a, complex b){
   complex t;
   t.real = a.real + b.real;
   t.imag = a.imag + b.imag;
   return t;
}
```

```c
int main(){
   complex a={3, 4};
   complex b={7, 8};
   complex x;
   x = add(a, b);
   printf("(%f,%f)\n",x.real, x.imag);
   return 0;
}
```

Output:

```
(10.000000,12.000000)
```

Structures and Pointers

```
struct rec {
   int roll;
   char name[50];
   float cpi;
};
struct rec *ptr;
```

Once `ptr` **points to a structure variable, the members can be accessed as:**

```
ptr->roll;
ptr->name;
ptr->cpi;
```

The symbol `->` **is called arrow operator.**

Arrow operator used to access members of a structure through structure type pointer.
`ptr->member` **is a shortcut for** `(*ptr).member`**.**

Structures and Pointers

```
typedef struct {
   float re, im;
} complex;
void add(complex *a, complex *b, complex *c){
   c->re = a->re + b->re;
   c->im = a->im + b->im;
}
int main(){
   complex a, b, c;
   scanf("%f%f",&a.re, &a.im);
   scanf("%f%f",&b.re, &b.im);
   add(&a, &b, &c);
   printf("(%f,%f)\n",c.re, c.im);
   return 0;
}
```

Note

- When using structure pointers, we should take care of operator precedence
  - Member operator " . " has higher priority than dereferencing operator "*"
    - `ptr->roll` and `(*ptr).roll` mean the same thing
    - `*ptr.roll` will lead to error
  - The operator "->" enjoys the highest priority among operators
    - `++ptr->roll` will increment `roll` not `ptr`
    - `(++ptr)->roll` will increment the pointer to the next structure in an array and access `roll` in the next structure

Unions

- In a struct, space is allocated as the sum of the space required by its members
- In union, space is allocated as the union of the spaces required by its members
  - We use union when we want only one of the members, but do not know which one
- Suppose that we wish to store the height of a student in one of the following formats
  - In FPS system, it is a string like 5’10” (use char array of size 8)
  - In CGS system, it is an integer like 178 (4 bytes are needed)
  - in MKS system, it is a floating point number 1.78 (4 bytes needed)
  - If we use a structure with all these members, total 16 bytes will be needed
  - A union will use only max(8,4,4) = 8 bytes of space
  - Additional flag needs to be stores to which representation it is

Union example

```
typedef struct {
   char name[20];
   float cpi;
   char htype;
   union {
      char fps[8];
      int cgs;
      float mks;
   } height;
} stud;
```

Output:

```
Union size: 8
Enter type of
height (f/c/m):f
Enter height:2'10"
2'10"
```

```
int main(){
   stud s;
   printf("Union size: %lu\n",sizeof(s.height));
   printf("Enter type of height (f/c/m):");
   scanf("%c",&s.htype); printf("Enter height:");
   if(s.htype == 'f') scanf("%s",s.height.fps);
   else if(s.htype == 'c') scanf("%d",&s.height.cgs);
   else if(s.htype == 'm') scanf("%f",&s.height.mks);
   switch (s.htype){
      case 'f': printf("%s\n",s.height.fps); break;
      case 'c': printf("%d\n",s.height.cgs); break;
      case 'm': printf("%f\n",s.height.mks); break;
      default: printf("Unknown\n"); break;
   }
   return 0;
}
```

Practice problems

- Define a structure to store points on 2D plane. Define a structure to store a rectangle. Assume that the sides of the rectangle are parallel to X and Y axis. Therefore you need to store bottom-left and top-right corners of a rectangle. Additionally, you can have an id (int) for each rectangle. Define an array of such rectangles. Populate the array with some random rectangles. Given the ids of two rectangles, determine whether those are overlapping. Find out the smallest rectangle that covers all the rectangles present in the array.
- Suppose you need to maintain records of student information. For each student, you need to store roll numbers (string) and grades for 6 subjects. Grades for a subject can be AA, AB, BB, BC, CC, CD, and DD. Here, AA refers to 10 and DD refers to 4. Similarly for other grades. Assume that the credit for 6 subjects are as follows: 4, 3, 5, 2, 1, 3. Display the whole record sorted in terms of (a) roll number, (b) cpi.